DELIVERABLE

Project Acronym: Europeana Cloud

Grant Agreement Number: 325091

Project Title: Europeana Cloud: Unlocking Europe's Research via The Cloud

Deliverable 5.2. Practical User Guide (Handbook for the Europeana Cloud participants)

Revision: 1.0

Authors: **Victor-Jan Vos, Europeana**
Pavel Kats, Europeana
Lucas Anastasiou, The Open University

Project co-funded by the European Commission within the ICT Policy Support Programme		
Dissemination Level		
P	**Public**	

Revision History

Revision	Date	Author	Description
0.5	27-8-2016	Lucas Anastasiou	DPS Plugin Tutorial
1	18-4-2016	Pavel Kats	Initial Version, aggregated tutorials
2	20-4-2016	Victor-Jan Vos	Revision
3	22-4-2016	Victor-Jan Vos	Final

Statement of originality:

This deliverable contains original unpublished work except where clearly indicated otherwise. Acknowledgement of previously published material and of the work of others has been made through appropriate citation, quotation or both.

D5.2 Europeana Cloud Practical User Guide

This deliverable describes the user documentation of the Europeana Cloud Shared Services. Since the shared services are meant to be used by developers, the practical user guide's intended audience are developers and stored on GitHub. In the future, Europeana will provide access to the documentation via Europeana Pro as well.

This documentation describes how aggregators' developers can make use of Europeana Shared Services for data partner and provide guidance for all features and functionalities of the service. It functions as a tutorial for uptake of the wider use of the services.

The first tutorial for a wide developer audience. It covers Europeana Cloud's data model and the Cloud API, which is what a typical developer needs to start using the services. So it is intended for developers in partner institutions - aggregators, data providers, re-users, and does not require any knowledge beyond the standard toolkit of today's developer.

The tutorial is maintained on Github:

https://github.com/europeana/Europeana-Cloud/wiki/Europeana-Cloud-API

The second tutorial is for developers who wish to develop new services (i.e. a Data Partner Service plugin) using Europeana Cloud Shared Services. It goes more in-depth to technology, demands acquaintance with the principles of distributed systems and distributed processing and uses a very specialised language. It is intended for a small audience of highly-skilled developers, interested in being intimately involved with the infrastructure. The tutorial is shown below.

DPS plugin development tutorial

A quick look in the processing architecture

Europeanna Cloud bases its processing infrastructure on the distributed computation framework Apache Storm[1]. Storm enables users to create their custom processing applications by utilising a series of application interfaces and combining them in "topologies".

...

[1] https://storm.apache.org/

Creating a new processing node (spout or bolt)

To create a new node (apart the given ones) to implement your custom business logic you need to extend the abstract class AbstractDpsBolt (found in DPS service commons module under eu.europeana.cloud.service.dps.storm package).

This means your Bolt code should look something like:

```
package com.mycompany.dps_tutorial;

import eu.europeana.cloud.service.dps.storm.AbstractDpsBolt;
import eu.europeana.cloud.service.dps.storm.StormTaskTuple;

/**
 *
 * @author lucasanastasiou
 */
public class MyBolt extends AbstractDpsBolt {

    @Override
    public void execute(StormTaskTuple t) {
        //
        // your code goes here
        //
    }

    @Override
    public void prepare() {
        //
        // your code goes here
        //
    }
}
```

As you can see there are two methods that you need to implement. The first one is void prepare() where you put everything that needs to be initiated before the actual processing takes place. Here would be a good place for example to establish a database connection or to initialise an external service. Bear in mind that if nothing needs to be initialised it can be left empty.

The second method is void execute(StormTaskTuple t). Here goes everything that implements the actual processing. It takes as an input a StormTaskTuple which is a Storm tuple that is aware of what DPS task is currently executing. Note that function is a void so it does not return anything but if you want to send something to the next bolt in line you should make use of the

outputCollector. So after you finish with processing you shall construct a new StormTaskTuple and emit it to the next bolt using the OutputCollector.

So your execute(StromTaskTuple input) function would basically look like

```
@Override
    public void execute(StormTaskTuple input) {
       // step 1 - get data from tuple
       Map<String,String> parameters = input.getParameters();
        inputData = parameters.get("key-of-your-input-data")

        // step 2 - do something with these data
       // implement your algorithm here

       // step 3 - construct an output tuple to emit to next bolt
       StormTaskTuple outputTuple = new
StormTaskTuple(taskId,outputData, outputParameters);
        _collector.emit("stream-to-next-bolt",outputTuple);
        _collector.ack(input);
    }
```

What features are available

As explained in the previous section Storm topologies comprise of spouts and bolts connected in a directed graph. Europeana Cloud offers some ready to use components for developers to combine and build their own topology implementing their own business logic. Such nodes are:

- Read File Bolt
- Write Record Bolt
- Read Dataset Bolt
- Store File as a new Representation Bolt
- Progress Bolt
- Parse Task Bolt

#TODO add a short description of each bolt's function

Combining nodes to a topology

Assuming you have all your spouts and bolts next step would be to meaningfully combine them to implement the workflow you have in mind.
To define the topology you would need to use TopologyBuilder class. So first you instantiate a new TopologyBuilder by

```
TopologyBuilder builder = new TopologyBuilder();
```

and then you set the processing nodes for this topology by using the setSpout and setBolt methods. Eventually you topology building block would look like:

```
TopologyBuilder builder = new TopologyBuilder();
builder.setSpout("name-of-spout", new MySpout(), 5);
builder.setBolt("name-of-bolt-1", new MyBolt(arg1,arg2),5);
builder.setBolt("name-of-bolt-2", new MyOtherBolt(arg3,arg4),5);
```

Method setSpout/setBolt takes 3 arguments:
id: a name to identify your node
spout/bolt: your node implementation (it needs to implement the IRichSpout / IRichBolt interface)
parralelism_hint: a number to denote the initial number of threads of this component

There are various ways to connect the defined bolts i.e. to explicitly declare the “edges” of your topology, the simplest way is by streams groupings. When defining the bolt in your builder, to group bolt1 with the spout you would do:

```
builder.setBolt("name-of-bolt-1", new MyBolt(arg1,arg2),5)
      .shuffleGrouping(“name-of-spout”);
```

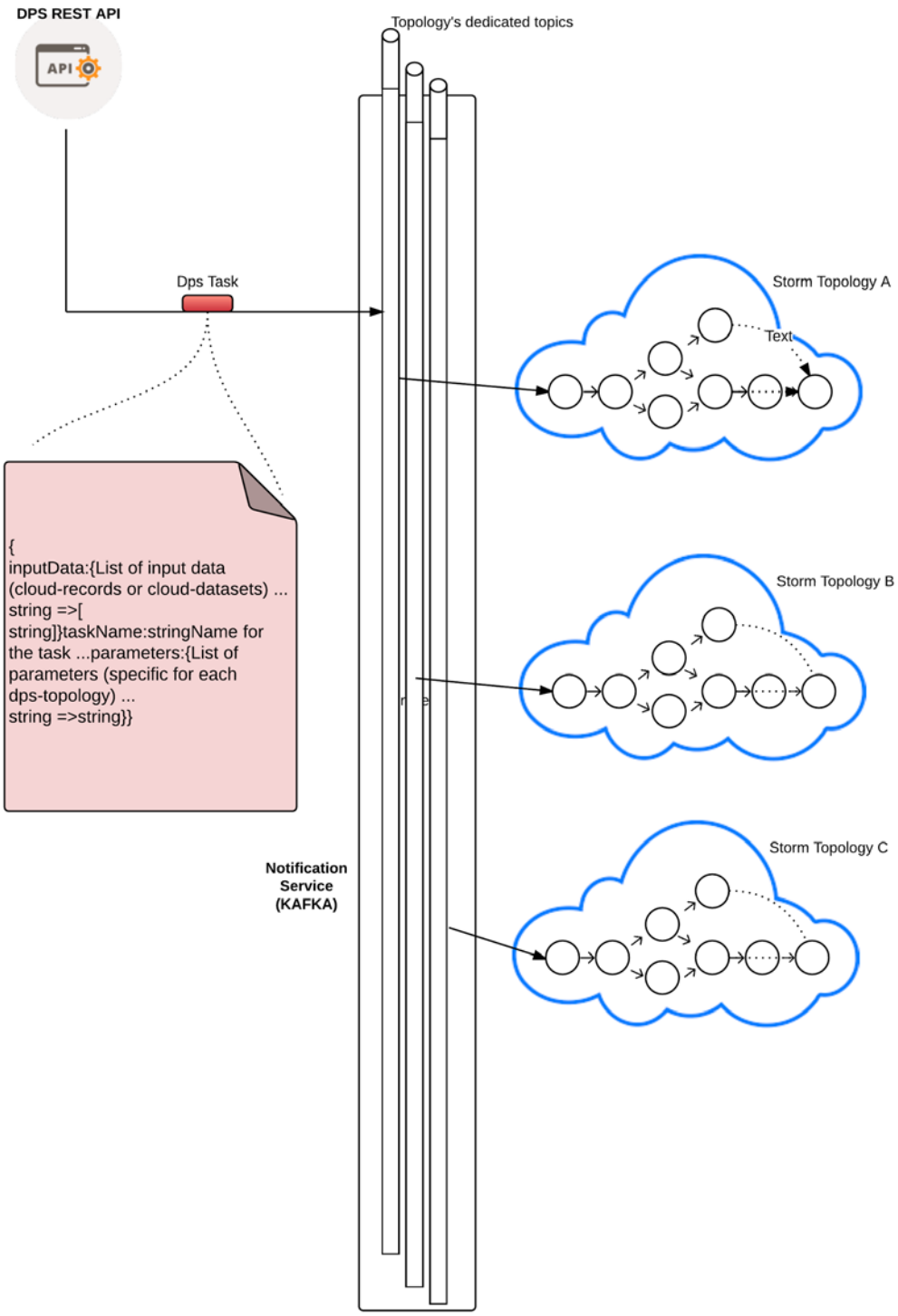
So by simply using the shuffleGrouping(id) method you declare that bolt1 is connected to the spout. And to connect bolt2 to bolt1:

```
builder.setBolt("name-of-bolt-2", new MyOtherBolt(arg3,arg4),5)
      .shuffleGrouping(“name-of-bolt-1”);
```

#TODO mention shuffleGroupings with 2 args (boltID, streamID)

So by the above small code pieces you have created a topology like figure 1:
#TODO
<figure of 3 circles : spout -> bolt1 ->bolt2>

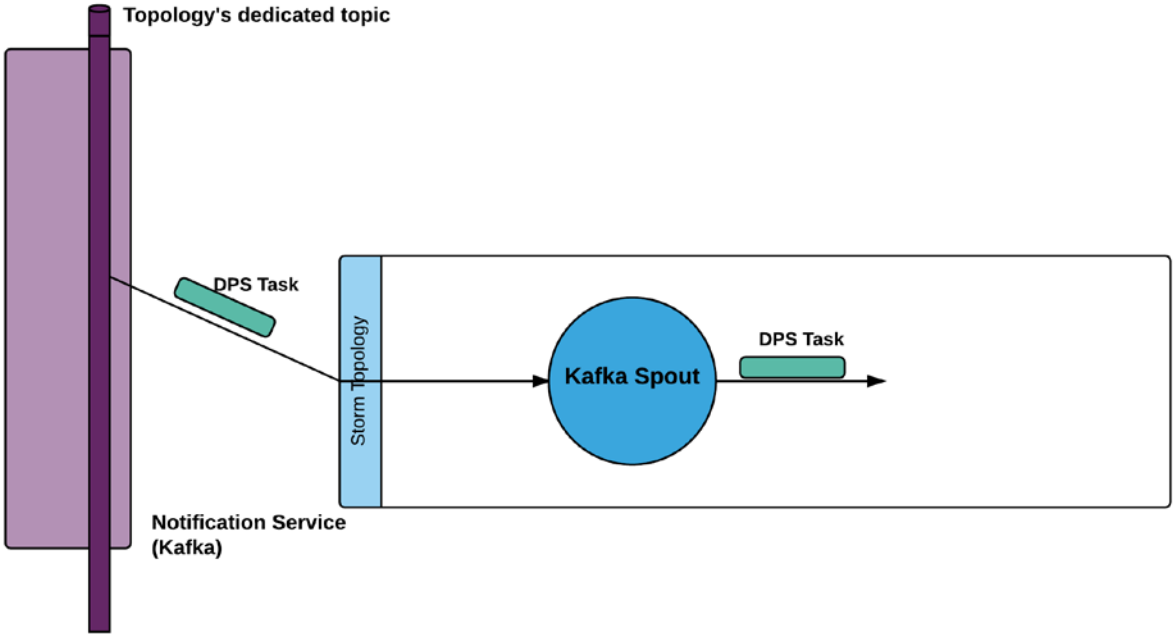
Running your tologies:
In Local mode
Deploy in Storm cluster
#TODO end


DPS REST API
API
Topology's dedicated topics
Dps Task
Storm Topology A
Text
{
inputData:{List of input data
(cloud-records or cloud-datasets) ...
string =>[
string]}taskName:stringName for
the task ...parameters:{List of
parameters (specific for each
dps-topology) ...
string =>string}}
Storm Topology B
Storm Topology C
Notification
Service
(KAFKA)

A plugin to convert .png to .jpeg

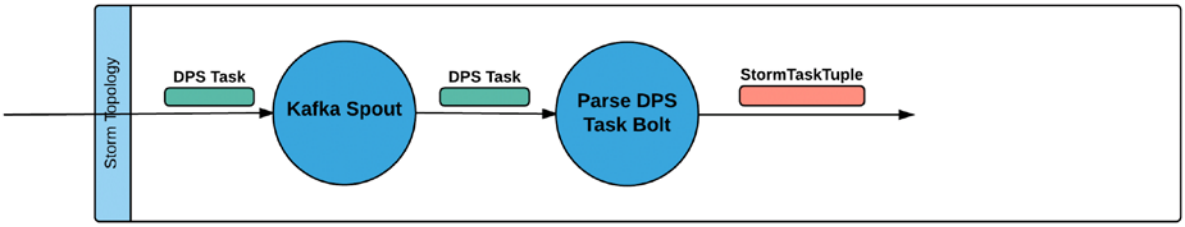
So far we explained how to use Storm in general to build arbitrary topologies but we also showed some of the ready-to-use features of eCLoud Data Processing Service. We would now combine everything to build a simple plugin to convert images stored in eCLoud records from one format (e.g. .png) to other (e.g. jpeg) and store the converted image as a new representation in the record. We would also show how this topology shall be used in the eCloud environment (how to start such processing task, how to monitor progress, how to see a final report of the execution of the task).

Your topology starts with a spout!



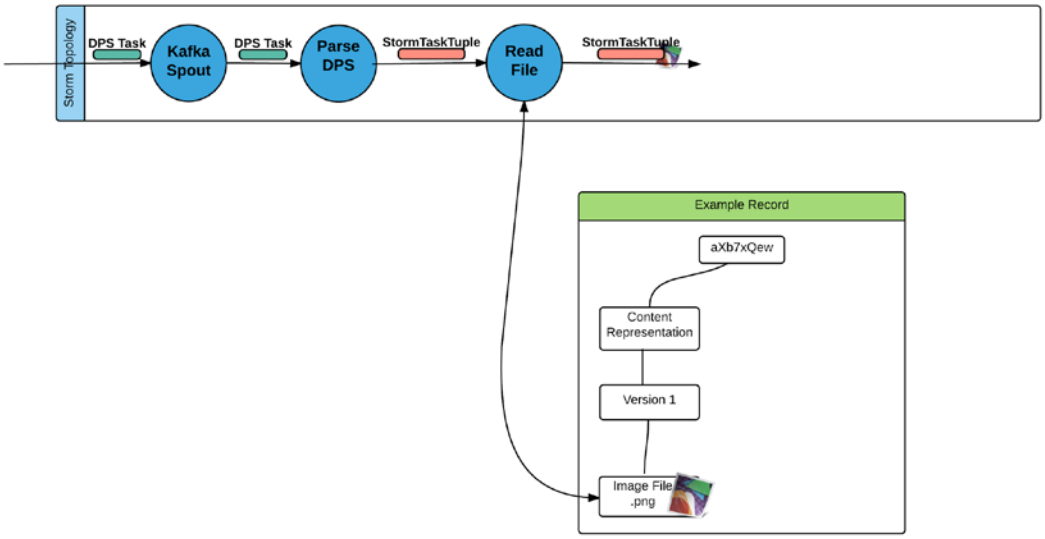
The initial node is of course a spout that is only responsible to connect to the DPS Rest API and “listen” for any incoming DPS Tasks. This connection is implemented through the notification service (Apache Kafka) where DPS Rest APi acts as a producer, each topology has its own dedicated kafka topic and the spout acts as a consumer of kafka messages. So it receives a kafka message which contains the DPS task (in a serialised form) and emits this task (as a tuple) to the next node of the topology.

Adding a bolt



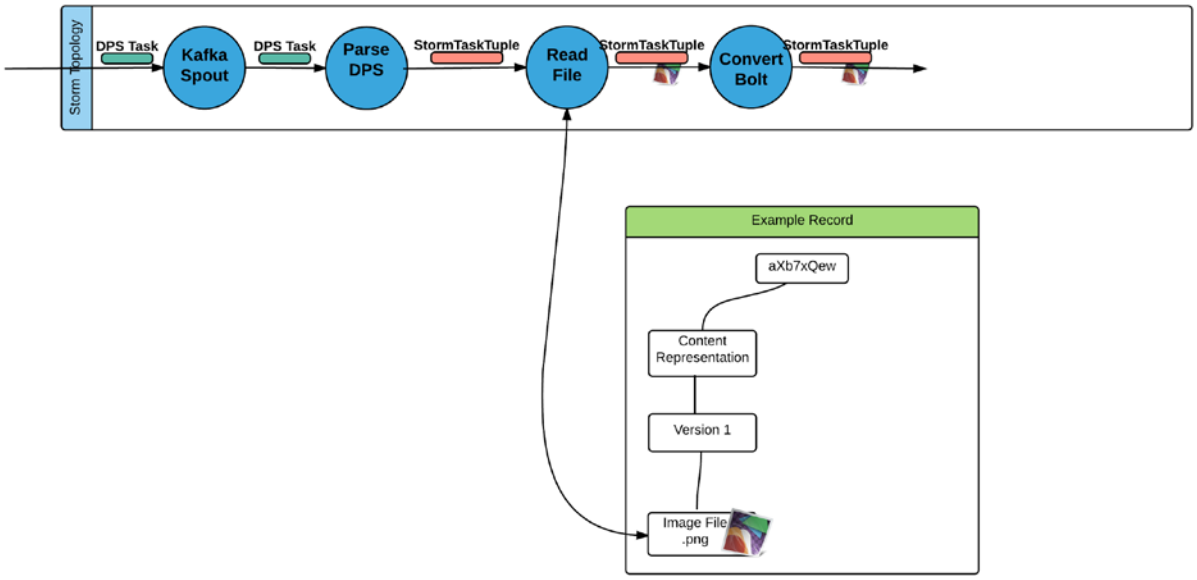
The second node of the topology would be a ParseDPSTask bolt i.e. a bolt that would identify what are the inputData and what are the parameters enclosed in the serialised DPS Task and convert them into a meaningful to Storm data structure: StormTaskTuple (a storm tuple that is aware of the dps task that it is part of). Another use of this bolt is to “route” tuples to specific streams by task name (in case you are building a more complex topology than the one we are showing here). Also when you instantiate this bolt you can also define what are the required parameters for your topology.

Read the image file from MCS



Next bolt would be a bolt to read the image file from MCS. Having the inputData enclosed in the StormTaskTuple we have received from the previous bolt, we can retrieve the resource by its URI and embed its binary data to the tuple that we are emitting to the next bolt

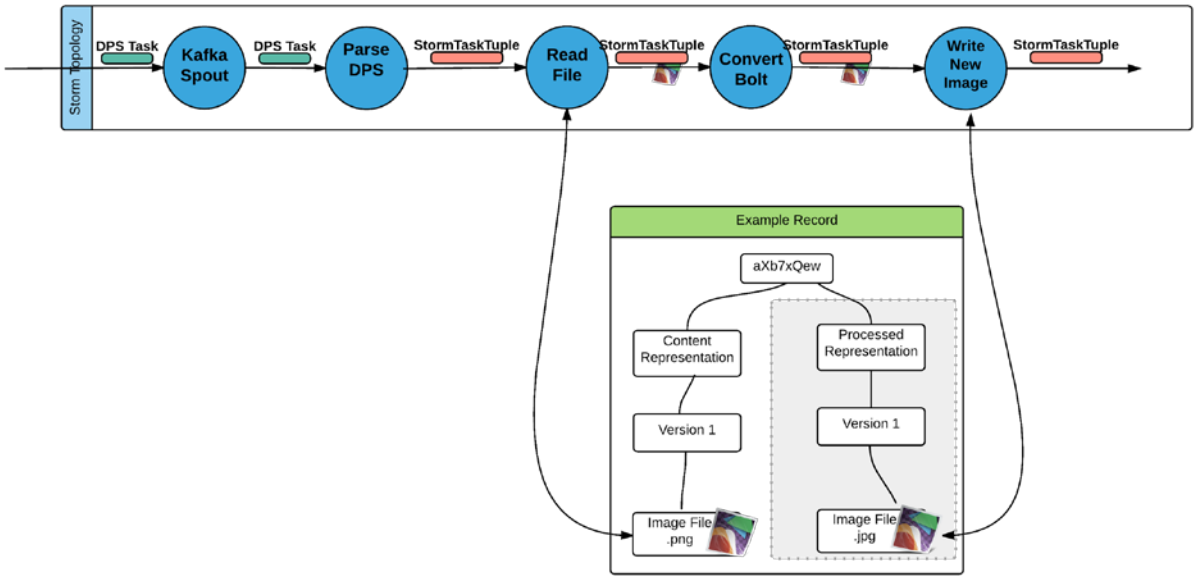
Convert image bolt



Now that we have the image (as binary input stream enclosed in the storm tuple) we can proceed to do the actual processing : transcoding the image from the input .png format to .jpeg format. To do so we use the standard java library ImageIO:

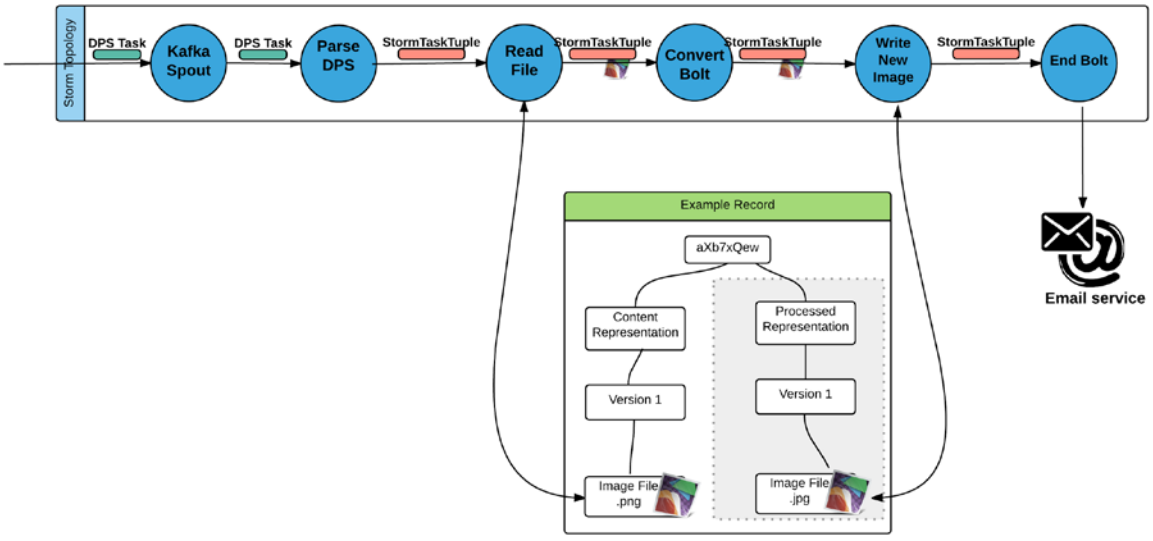
```
try {
            // fetch the image from the input tuple
            ByteArrayInputStream inputImage =
t.getFileByteDataAsStream();
            BufferedImage bufferedImage = ImageIO.read(inputImage);
            OutputStream os = new ByteArrayOutputStream();
            // transcoding to jpeg
            ImageIO.write(bufferedImage, "jpg", os);

            t.setFileData(os.toString());
            //emitting to the next bolt
            outputCollector.emit("stream-to-next-bolt",
t.toStormTuple());

        } catch (IOException ex) {
}
```

Store back as a new representation



Using the StoreFileAsRepresentationBolt we store the converted image as a new representation in the record that the initial representation exists.

End processing



We finish our topology by an EndBolt. This bolt can be optionally used to send an email notification to the end user that the task finished execution (mind that in this example topology we are dealing only with one resource so finishing the processing of one tuple is equivalent to finishing the execution of the task but in the case we are dealing with many resources per task

this can be tricky to identify and can only be achieved by the introduction of the Notification bolt we will see later).

Putting everything together

To build our topology we use TopologyBuilder again:

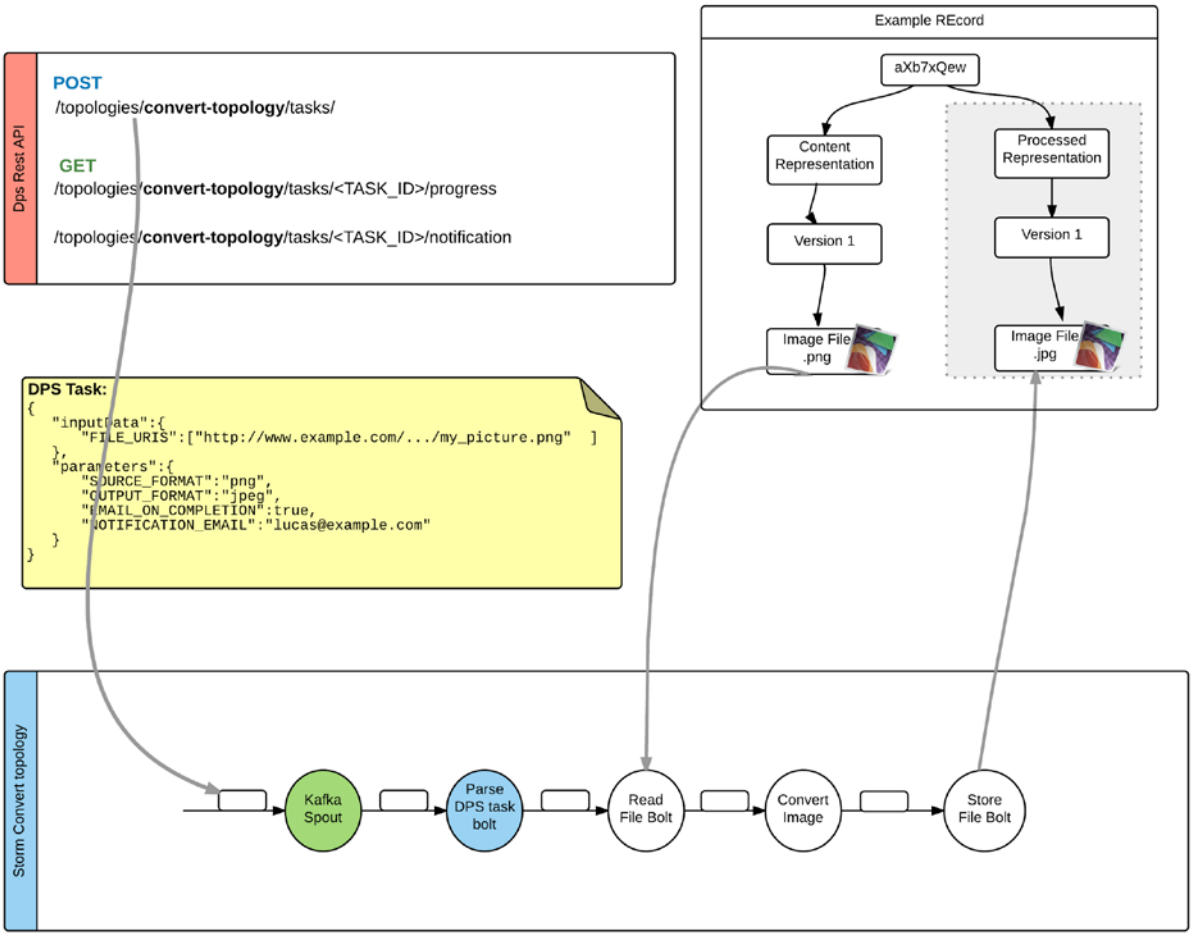
```
TopologyBuilder builder = new TopologyBuilder();

        // entry spout
        BrokerHosts brokerHosts = new ZkHosts(ZOOKEEPER_LOCATION);
        SpoutConfig spoutConf = new SpoutConfig(brokerHosts,
KAFKA_TOPIC, ZK_ROOT, ID);
        builder.setSpout("KafkaSpout", new KafkaSpout(spoutConf), 1);

        //bolt 1
        builder.setBolt("ParseDpsTask", new ParseTaskBolt(), 1)
                .shuffleGrouping("KafkaSpout");

        //bolt 2
        builder.setBolt("RetrieveFile", new
ReadFileBolt(ECLOUD_MCS_ADDRESS, ECLOUD_MCS_USERNAME,
ECLOUD_MCS_PASSWORD), 1)
                .shuffleGrouping("ParseDpsTask");

        //bolt 3
        builder.setBolt("ConvertBolt", new ConvertBolt(), 1)
                .shuffleGrouping("RetrieveFile");

        //bolt 4
        builder.setBolt("StoreBolt", new
StoreFileAsRepresentationBolt(ECLOUD_MCS_ADDRESS, ECLOUD_MCS_USERNAME,
ECLOUD_MCS_PASSWORD), 1)
                .shuffleGrouping("ConvertBolt");

        //bolt5
        builder.setBolt("EndBolt", new EndBolt(),
1).shuffleGrouping("StoreBolt");
```

How to invoke your plugin

After you have deployed the storm topology to the storm cluster you can interact with it via the DPS REST API. To submit a new task you would do:

```
$> curl -u admin:admin -XPOST
http://www.example.com/topologies/convert-topology/tasks -d '
{
      "task_name":"convert_image",
      "inputData":{
            "FILE_URIS":["http://www.example.com/.../my_picture.png"  ]
      },
      "parameters":{
            "SOURCE_FORMAT":"png",
            "OUTPUT_FORMAT":"jpeg",
            "EMAIL_ON_COMPLETION":true,
            "NOTIFICATION_EMAIL":"lucas@example.com"
      }
}'

$> {"taskId":"12345", "time":123456789010}
```

This POST request submits a DPS Task to the notification service and task shall soon be picked up by Storm's topology and start executing (usually after a few ms). End user only gets the generated task ID and the unix timestamp of the time of the submission.



What about progress?

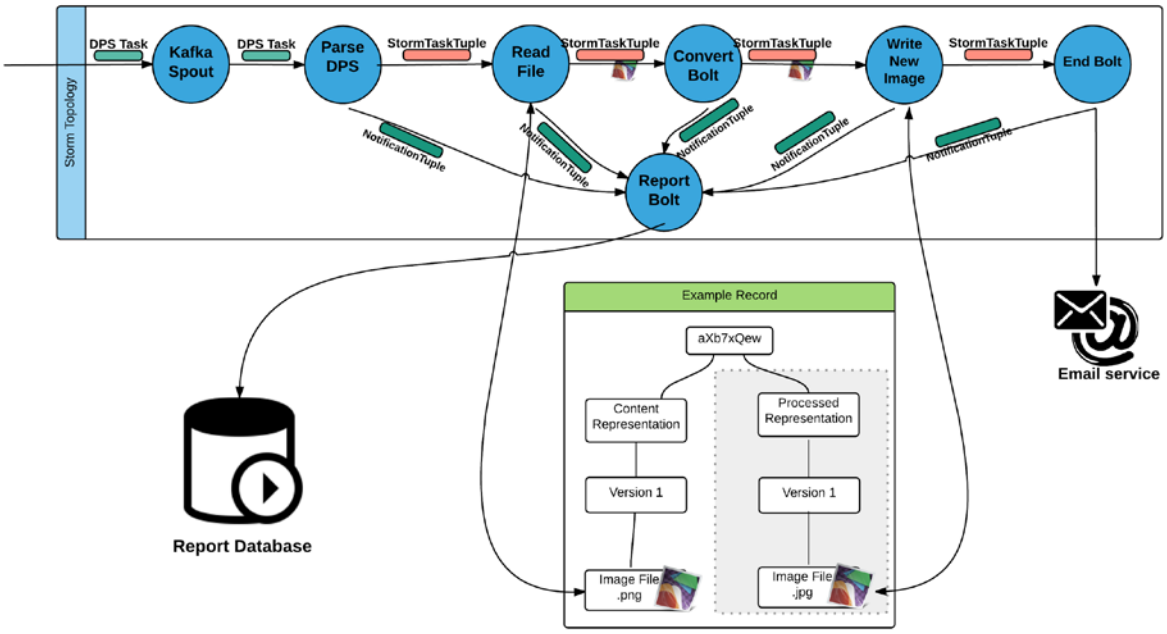
You might have noticed that so far there is no mechanism to capture the progress of this task. It may be a task containing just one resource to process but it can be easily modified to take as input many files or even a full dataset (of perhaps thousands records to process). In such case it would be very useful for the client executing the task to know the progress of the task.

Introducing the Notification bolt

As part of the ready-to-use bolts and spouts, eCLoud offers Notification bolt : a special bolt that can act as a monitor of the execution of tasks in each topology. In order to use you should connect this bolt to each existing bolt of your topology, for instance for the convert topology you should add in you TopologyBuilder:

```
 //notification bolt
        builder.setBolt("NotificationBolt", new
NotificationBolt("convert-topology", CASSANDRA_HOSTS, CASSANDRA_PORT,
CASSANDRA_KEYSPACE_NAME, CASSANDRA_USERNAME, CASSANDRA_PASSWORD), 1)
```

```
        .shuffleGrouping("ParseDpsTask", NOTIFICATION_STREAM)
        .shuffleGrouping("RetrieveFile", NOTIFICATION_STREAM)
        .shuffleGrouping("ConvertBolt", NOTIFICATION_STREAM)
        .shuffleGrouping("StoreBolt", NOTIFICATION_STREAM)
        .shuffleGrouping("EndBolt", NOTIFICATION_STREAM);
```



Every time you gather some information (successful process of a resource, some error, exception) you should create a NotificationTuple in the bolt that this update happens and send it to the Notification Bolt which in turn will update a database with this info. This report database will be later on queried by DPS Rest API via /progress and /notification calls to return report information on the status of the task. To ease this “notify” operation eCloud has various utility functions already implemented in each bolt (assuming you are extending AbstractDpsBolt):

- emitDropNotification(long taskId, String resource, String message, String additionalInformations)
- emitSuccessNotification(long taskId, String resource, String message, String additionalInformations
- emitErrorNotification(long taskId, String resource, String message, String additionalInformations)
- emitBasicInfo(long taskId, int expectedSize)

So in the current example convert-topology the convert bolt now looks:

```
@Override
    public void execute(StormTaskTuple t) {
```

```
        try {
            ByteArrayInputStream inputImage =
t.getFileByteDataAsStream();
            BufferedImage bufferedImage = ImageIO.read(inputImage);
            OutputStream os = new ByteArrayOutputStream();
            ImageIO.write(bufferedImage, "jpg", os);

            t.setFileData(os.toString());
            outputCollector.emit("stream-to-next-bolt",
t.toStormTuple());

        } catch (IOException ex) {
            // send notification of error
            this.emitErrorNotification(t.getTaskId(), t.getFileUrl(),
ex.getMessage(), "");
            return;
        }
        this.emitSuccessNotification(t.getTaskId(), t.getFileUrl(),
"Yey!", "");
    }
```

Checking the progress of a DPS task

Now that we have a progress mechanism implemented in our topology, if we wish to see the progress of the task then you do:

```
$> curl -u admin:admin -XGET
http://www.example.com/topologies/convert-
topology/tasks/12345/progress/
$> {"taskId":"12345", "expected_size":1,
"processed_tuples_count":0,”topology_name”:”convert-topology”}
```

Report of a DPS task

To get a full list of what tuples have been processed (or perhaps when the task is done to have a full report) you would do:

```
$> curl -u admin:admin -XGET
http://www.example.com/topologies/convert-
topology/tasks/12345/notification/
$> {"taskId":"12345","processed_tuples":[
```

```
{"resource_URI":"http://www.example.com/records/aXb7xQew/representa
tions/image/versions/0ewer2xqwl/files/image.png/",
"time":123456789012, state":"SUCCESS", "information":"",
"additional_information":""}
]}
```


Mind that if we were dealing with say a dataset that would have many resources associated with it the above call would look something like:


```
{"taskId":"12346","processed_tuples":[
{"resource_URI":"http://www.example.com/records/aXb7xQew/representatio
ns/image/versions/0ewer2xqwl/files/image.png/", "time":123456789012,
state":"SUCCESS", "information":"", "additional_information":""},
{"resource_URI":"http://www.example.com/records/ar45dakm/representatio
ns/image/versions/93naiqufnd/files/image.png/", "time":123456789015,
state":"ERROR", "information":"javax.imageio.IIOException: Can't read
input file!
Resource not found: C:\icon.gif
at javax.imageio.ImageIO.read(ImageIO.java:1301)
at connector.SystemTrayCreator.createImage(SystemTrayCreator.java:98)
at connector.SystemTrayCreator.create(SystemTrayCreator.java:36)
at connector.Start.main(Start.java:14)",
"additional_information":"Error while converting image"},
{"resource_URI":"http://www.example.com/records/2305ufaj/representatio
ns/image/versions/akas0111l2l3/files/image.png/", "time":123456789015,
state":"SUCCESS", "information":"", "additional_information":""}
]}
```


Note that in the case of the error tuple we have the full java exception in the information field detailing what went wrong. As developer you are free to decide what sort of information you would like to put in this field.